CMSC425 Midterm 2 Prep and practice

The midterm on May 8th will be in class, closed book, and similar to Midterm 1. There will be 5 to 6 questions, up to seven pages, with the first question short answer and the rest applications of the concepts. Questions from the homeworks are fair game, as are questions from lectures before spring break and the practice midterm exams from spring and fall 2018. Question on Unity will be limited, and based on what you should have learned in Project 1.

Possible concepts and questions include:
1. Metrics for best path on map
2. Navmesh process (R_D_P algorithm, triangulation)
3. Walkable terrain
4. Find paths on triangulated space
5. **Configuration spaces**
6. Quality of path
7. **C-obstacles**
8. **Minkowski sums**
9. Navmesh - grid, mulitresolution grid
10. Visibility graph
11. Medial axis
12. Randomized placement
13. Rapidly-expanded Random Trees (RRTs)
14. **L-system plus turtle**
15. **Fractal dimension**
16. Randomized and 3D L-systems
17. Particle systems
18. **Flocking**
19. Mandelbrot sets
20. Constructive solid geometry
21. Shading equation
22. Bump mapping
23. Polygonal meshes - basics, Euler's formula
24. **DECL data structures**
25. Perlin noise
26. **A***
27. Admissible heuristic
28. Multiplayer cheating attacks
29. Forbidden velocities for crowd motion
30. Curves and patches (linear, cubic, bilinear, Hermite cubic, matrix representation)

Practice questions

See examples from lectures:
Day 22, slide 37
Day 20, slides 31, 32, 33
Day 19, slide 39
Day 17, slide 35
Day 15, slide 24

Examples from previous semesters:
(Some used on homeworks already)
Homework 2, Spring 17: Questions 1b, c, d, (not e); 2; 3; not sure about 4.
Second Midterm, Spring 17: Questions 1a, b, (not c), d, e, f; 2; 3; 4
Homework 2, Spring 2016: Questions 1a, b, (not c), d; 2; 3; (not 4)

CMSC 425 : Spring 2017 Dave Mount

Homework 2

Handed out Thu, April 27 (and revised Mon, May 1). Due at the start of class Thu, May 4. Late homeworks will not be accepted (without prior approval), so turn in whatever you have done.

Problem 1. Short answer questions.

(a) In skinning, what is principal reason for binding a mesh vertex to multiple joints?

(i) Allows joints to rotate in multiple directions (like a neck) as opposed to just one (like the middle joint of a finger)

(ii) Enhanced efficiency when rendering very large meshes

(iii) Allows the mesh to deform smoothly as the joints rotate

(iv) All of the above

(b) In the Ramer-Douglas-Peucker algorithm for simplifying a polygonal curve, what is the criterion for selecting the next vertex to be added to the curve?

(c) How many dimensions are there in the configuration spaces for each of the following motion-planning problems. Justify your answer in each case by explaining what each coordinate of the space corresponds to.

(i) Moving a line segment in 2-dimensional space, which may be translated and rotated.

(ii) Moving a line segment in 3-dimensional space, which may be translated and rotated.

(iii) Moving a pair of scissors in 3-dimensional space, which may be translated, rotated, and may swing open and closed.

(d) What is the principal advantage of selecting a path that travels along the *medial axis* of the free-space?

(e) Which of the following aspects of human sound perception allow us to determine the location that a sound comes from? (Select all that apply.)

(i) Our auditory systems detect differences in the arrival times of sounds to each of our ears

(ii) Sound waves reflect off our upper body, head, and outer ears in different ways depending on the origin of the sound

(iii) People move their head purposively while listening to a sound to determine its origin

(f) In our description of the four elements of the boid model for flocking behavior (separation, alignment, avoidance, and cohesion), what is the purpose of *cohesion*, and how might it be implemented?

Problem 2. Consider the quadrilateral A and square B, shown in Fig. 1. The quadrilateral A's reference point is located at the origin, that is, $p_a = (0, 0)$, and the other vertices are at $(2, 0)$, $(1, 2)$, and $(0, 2)$. The square B's reference point is at $p_b = (3, 4)$ and its other vertices are at $(5, 3)$, $(6, 5)$, and $(4, 6)$. Object A can translate but not rotate.

The *configuration obstacle* of B with respect to A is the set of placements of A's reference point so that it overlaps B. Describe (draw clearly or explain with coordinates) the configuration obstacle of B with respect to A.

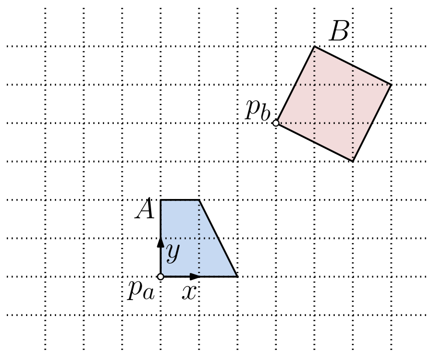


Figure 1: Problem 2.

Problem 3. In this problem we consider the performance of Dijkstra's algorithm and A* search on the graph shown below (see Fig. 2), where the objective is to compute the shortest path from s to t. Each edge (u, v) is undirected and is labeled with its associated weight $w(u, v)$.

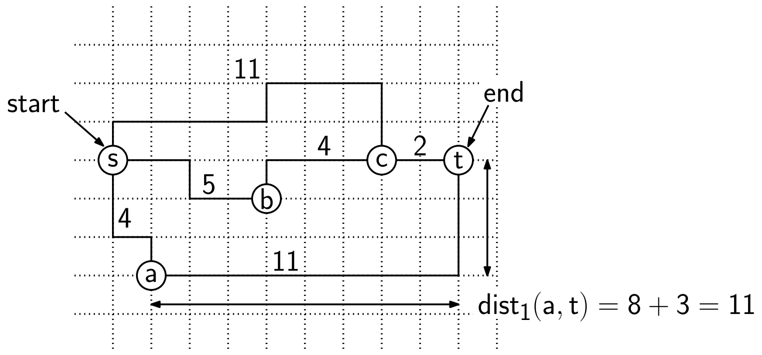


Figure 2: Problem 3.

For A* search we need to make use of a heuristic. To save you from dealing with square roots, as we did in the example from class, we will use the L_1 (also called Manhattan or checkerboard) distance between two points. It is defined to be the sum of the absolute values of the difference of the x and y coordinates of the points. For example, in the figure the L_1 distance between nodes a and t is $\text{dist}_1(a, t) = 8 + 3 = 11$. For A* search define the heuristic value for each node u to be L_1 distance from u to t. For example, $h(a) = 11$. (For this graph it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

(a) Trace the execution of Dijkstra's algorithm on this graph. For each node indicate the following: (1) list the nodes in the order in which they are processed, (2) whenever a node is processed, indicate which of its neighbors have had their d-values modified, when the algorithm terminates (that is, when t is considered for processing), indicate what the final d-values are for all the nodes. If there are ties for which node is to be processed next, select the node that is earliest in alphabetical order. (As an example, see Lecture 16.)

(b) Trace the execution of A* Search on this graph. Provide the same information as in (a), but also provide the A* f-values for each node that is processed (recall that $f(u) = d[u] + h(u)$).

(c) Remark on the differences (if any) in the length of the path generated and the differences (if any) in the efficiency between these two algorithms.

Problem 4. One of the options that Unity (and other game engines) offer when dealing with Navigation Meshes is the ability to specify the size of the moving agent. Suppose that the moving agent is a circular disk of radius r (see Fig. 3(a)). One way to deal with this size is to "shrink" the accessible domain by this radius before triangulating the domain. In this problem, we will explore a different method. (This method has some additional advantages, since we can change the value of the radius without recomputing the navigation mesh.)

Here is the idea. Suppose that we have already computed a navigation mesh of the *entire* accessible domain (no shrinking). The mesh is represented by a triangulation T, which is stored using a DCEL data structure. The mesh contains two types of faces: the triangles of the navigation mesh, and obstacle faces. The obstacle faces are not required to be triangles. In order to identify what type of face you have, each face f of the DCEL has an additional field isObstacle. If f.isObstacle is true, then this face is an obstacle. Otherwise, it is part of the navigation mesh. (For example, in Fig. 3(a), the two shaded faces, the external face and the inner triangle, are both obstacles.) Throughout, we will assume that every vertex of the navigation mesh is adjacent to one obstacle face, and two obstacle edges.

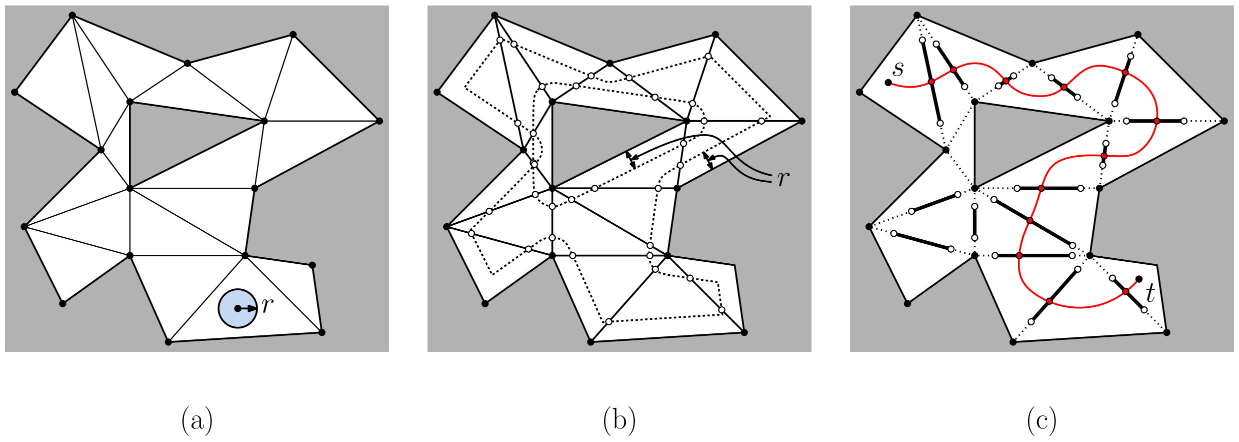


Figure 3: Problem 4.

Given that our moving agent is a circular disk of radius r, to determine where its center point may navigate, we shrink the accessible domain by a distance of r. (This is shown in the dashed lines of Fig. 3(b).) This is called an *offset contour*. Rather than storing the entire offset contour, we will just store where it intersects the edges of the triangulation. (These are shown as white dots in Fig. 3(b).) Observe that the agent can safely move from one triangle to a neighboring triangle if the edge between these triangles contains a *window*, that is, a line segment that lies entirely within the portion of the mesh that is beyond the offset contour. (These window segments are shown as heavy lines in Fig. 3(c). Two edges of the triangulation in the figure are so short that they do not have a window, and therefore the agent cannot travel across these edges.)

In this problem, we will investigate how to compute these windows and use them to compute paths in that navigation mesh that provide a clearance of r from the obstacle boundaries. **Note:** After writing the problem, I realized that there are conditions under which an edge may have multiple windows. To simplify matters, let's assume that this never happens. For example, I believe that if all the triangles of the navigation mesh are acute, each edge can have at most one window.

(a) Let e be any half edge of the navigation mesh such that e.left.isObstacle (see Fig. 4(a)). Using the operations of a DCEL, explain how to obtain the half edges e' and e'' that immediately precede and follow e, respectively, in counterclockwise order around the obstacle face. Also, explain how to obtain the vertices a and b at e's origin and destination, respectively. The operations should all take $O(1)$ time.

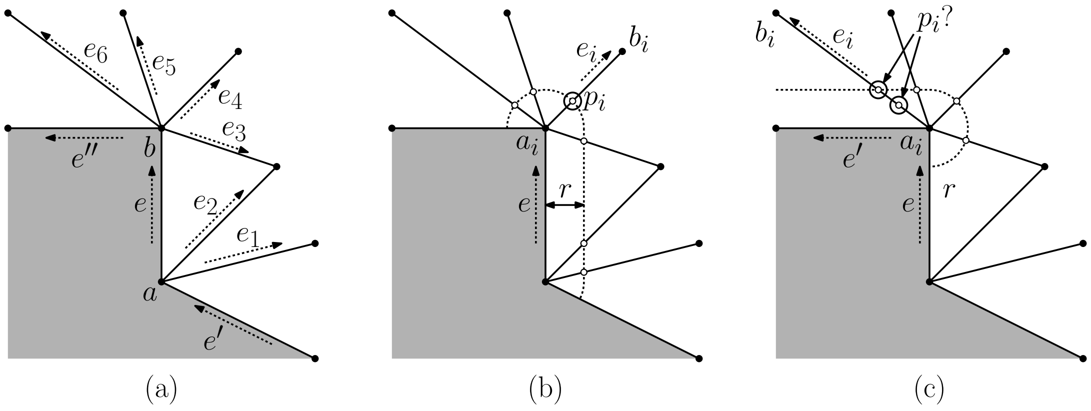


Figure 4: Problem 4 (continued).

(b) Using the operations of a DCEL, present a code fragment that returns a list L containing the half edges $\langle e_1, \ldots, e_k \rangle$ that are incident to a and b, not including edges that border the obstacle that is incident to e. (For example, in Fig. 4(a), L consists of $\langle e_1, \ldots, e_6 \rangle$.) These half edges should all be directed outwards from a and b, and the list should be computed in time proportional to the number of edges in the list.

(c) For any half edge $e_i \in L$, let a_i and b_i denote its origin and destination, respectively (see Fig. 4(b)). Given distance $r \geq 0$, explain (using the geometrical operations discussed in class) how to compute the point p_i along this edge that is at distance r from the obstacle edge e. (Consider only the edge e. The other incident edges e' and e'' will be considered in the next parts.)

Hint: I found it helpful for solving parts (d) and (e) to represent the point p_i as $a_i + \alpha \vec{v}_i$, where $\vec{v}_i$ is a vector aligned with the line segment $\overline{a_i b_i}$, but this is not a requirement. There are cases, depending on whether $a_i = a$ or $a_i = b$, and whether the angle between the (undirected) edges e_i and e is acute or obtuse.

(d) The point p_i computed in part (c) considers only one of the two obstacle edges incident on a_i. Assuming that we have applied part (c) to both of the edges incident to a_i, explain how to compute which of these two points are to be used for the purposes of computing the window along this edge. (For example, in Fig. 4(c) we show two possible points p_i, one computed for edge e and the other for e'. How would you determine which of the two defines the endpoint of the window?)

(e) Given your answer to (d) and under our assumption that each edge has at most one window, for each triangulation edge (a_i, b_i), we have two window endpoints. One point p_i is at distance r from the obstacle incident to a_i, and the other point q_i is at distance r from the obstacle incident to b_i. Explain (using the geometrical operations discussed in class) how to determine whether the window segment for this edge is nonempty.

By the way, here is an explanation of how to complete the path computation, assuming you have successfully solved parts (a)–(d). Given a source point s and a destination point t, we first verify that both are at distance at least r from the nearest obstacle (otherwise there clearly is no path). Assuming so, we determine whether there is a path between them by computing the dual graph of the triangulation, where we keep only those dual edges whose corresponding windows are nonempty. Then, we determine whether the triangles containing s and t are connected by a path in this graph. Such a path passes from one triangle to the next through a window. It is not hard to show that if two edges of a triangle have nonempty windows, there exists a path of clearance r from one edge to the other. (You do not need to explain how to do this.)

Challenge Problem: In Problem 4, I observed that it is possible for a triangulation edge to have multiple windows. Draw an example to illustrate how this can happen. For a real challenge, prove that if all the triangles of the navigation mesh are acute, for any value of $r \geq 0$, each edge has at most one window.

CMSC 425 : Spring 2017 Dave Mount

Second Midterm Exam

This exam is closed-book and closed-notes. You may use two sheets of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (30 points, 3-8 points each) Short answer questions. Explanations are not required, but may be given for partial credit.

(a) The Unity mechanim animation system has a feature that can set the joint angles of a humanoid model to cause it to turn its head to face a particular direction. (Really!) This would best be described is an example which of the following animation techniques (select one):

- (i) Keyframe animation
- (ii) Motion capture
- (iii) Inverse kinematics

(b) Our proposed algorithm for triangulating the walkable region in the construction of navigation meshes repeatedly cut off the *ear* such that the cutting edge has minimum length. What is an *ear* of a simple polygon? What is the reason for favoring short cutting edges?

(c) List one advantage and one disadvantage of using the *potential-field* approach to computing paths.

(d) In our description of the four elements of the boid model for flocking behavior (separation, alignment, avoidance, and cohesion), what is the purpose of *alignment*, and how might it be implemented?

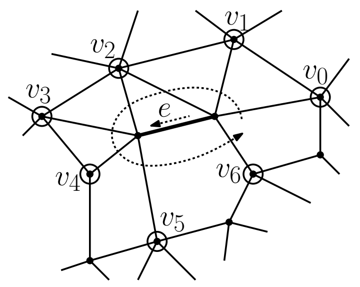
(e) Behavior trees have two types of task nodes, sequences and selectors. In a *sequence node*, its children are evaluated from left to right, and each returns either *success* or *failure*. Under what circumstances does the sequence node itself return success?

(f) Repeat (e), but this time for a *selector node*.

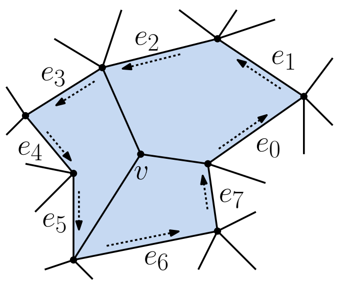
Problem 2. (25 points) Answer the following questions assuming that you are given a planar subdivision (i.e., cell complex of a 2-manifold) represented as a DCEL.

(a) Present a procedure (in pseudocode) that, given a half edge e of the DCEL, returns a list L consisting of the vertices that are adjacent to either of e's endpoints. The vertices should be listed in *counterclockwise* order about e. The list can start with *any* vertex, and *duplicates* are allowed.

For example, given the example shown in Fig. 1(a), the list $L = \langle v_0, v_1, \ldots, v_6 \rangle$ would be one valid result (as would any cyclical shift of this sequence). Your procedure should run in time proportional to the length of the output. (Hint: The answer is simpler if you choose the starting point carefully.)



(a)



(b)

Figure 1: Problem 2.

(b) Given vertex v in a cell complex of a 2-manifold, the *link* of v is defined to be the edges that bound the faces that are incident to v, excluding the edges that are incident to v itself. (For example, in Fig. 1(b), the link of v consists of the edges $\langle e_0, e_1, \ldots, e_7 \rangle$.) Present a procedure (in pseudocode) that, given a vertex v of the DCEL, returns a list L consisting of the half edges of v's link. The half edges should be directed in *counterclockwise* order about v and the list should also be ordered in the same way. The list can start with any half edge of the link. Your procedure should run in time proportional to the length of the output.

Problem 3. (20 points) Consider the collection of shaded rectangular obstacles shown in the figure below, all contained within a large enclosing rectangle. Also, consider the triangular robot, whose reference point is located at a point s. (You may take s to be the origin.)

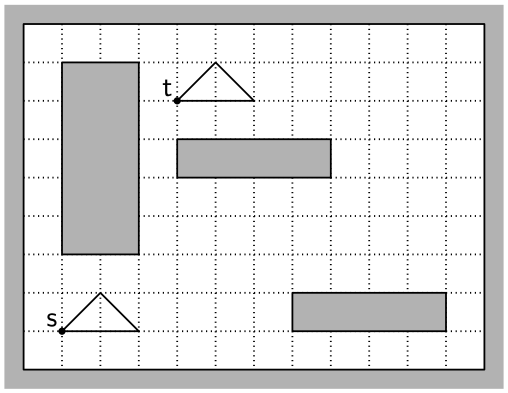


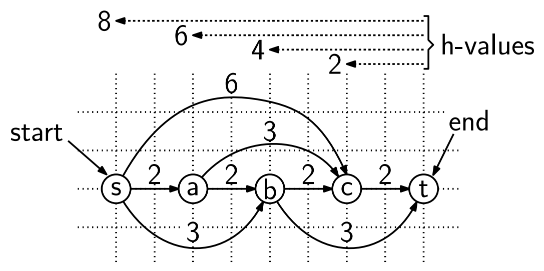
Figure 2: Problem 3.

(a) Draw the C-obstacles for the three rectangular obstacles, including the C-obstacle from region lying outside the large enclosing rectangle.

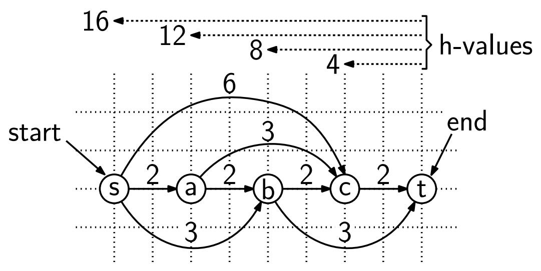
(b) Either draw an obstacle-avoiding path for the robot from s to t, or explain why it doesn't exist.

Problem 4. (25 points) In this problem, we will consider A* search under both admissible and inadmissible heuristics. *Please use the version of A* given in class.*

Consider the graph shown in Figure 3. For each node u, define $\text{dist}(u, t)$ to be the *straight-line distance* from u to t. For example, $\text{dist}(s, t) = 8$ and $\text{dist}(c, t) = 2$.



(a)



(b)

Figure 3: Problem 4.

(a) Suppose that we take the *admissible heuristic* $h(u) = \text{dist}(u, t)$ (see Fig. 3(a)). (For example, $h(s) = 8$, $h(c) = 2$ and $h(t) = 0$.) Trace the execution of the A* algorithm on this graph using s as the start and t as the destination. In particular:

- list the nodes as they are processed, indicating the values of $d[u] + h(u)$
- whenever a node is processed, indicate how the d-values of its neighbors are updated

At the end (when t is processed), show the final d-values are for all the nodes.

(b) Suppose that we take the *inadmissible heuristic* $h(u) = 2 \cdot \text{dist}(u, t)$ (see Fig. 3(b)). (For example, $h(s) = 16$, $h(c) = 4$ and $h(t) = 0$.) Repeat (a) but using this different value of h.

(c) Did the algorithm produce the correct answer in part (b)? Explain briefly.

A* Search

```
A-Star(G, s, t) {
  foreach (node u) {                     // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered           // distance to source is 0
  repeat forever {                       // go until finding t
    let u be the discovered node that minimizes d[u] + h(u)
    if (u == t) return d[t]              // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))  // update d[v]
        mark v discovered
      }
      mark u finished                    // we're done with u
  }
}
```

CMSC 425:Spring 2016 Dave Mount

Homework 2

Handed out Tue, May 3. Due at the start of class Tue, May 10. Late homeworks will not be accepted (without prior approval), so turn in whatever you have done.

Problem 1. Short answer questions.

(a) Consider the design of a decision-making AI system based on *behavior trees*. Give an example of a task involving multiple decisions where a *sequence task* would be appropriate.

(b) Given the same scenario as (a), give an example of a task involving multiple decisions where a *selection task* would be appropriate.

(c) List one advantage and one disadvantage of the use of *potential-field navigation* as a means for computing paths in a game.

(d) When computing navigation meshes, we applied a step that simplified the polygonal region that modeled the walkable area. (That is, we approximated this region by eliminating vertices.) What was the principal reason for doing this simplification? What would be the danger of excessive simplification (removing too many vertices)?

Problem 2. Consider the triangle a and polygon b, shown in Fig. 1. (Triangle a's reference point is located at the origin, that is, $p_a = (0, 0)$ and b's reference point is at $p_b = (2, 4)$.)

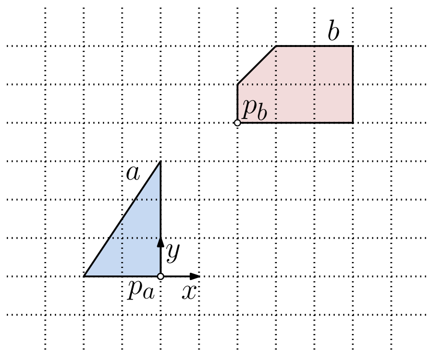


Figure 1: Problem 2.

The *configuration obstacle* of b with respect to a is the set of placements of a's reference point so that it overlaps b. Describe (draw clearly or explain with coordinates) the configuration obstacle of b with respect to a.

Problem 3. In this problem we consider the performance of Dijkstra's algorithm and A* search on the graph shown below (see Fig. 2), where the objective is to compute the shortest path from s to t. Each edge (u, v) is undirected and is labeled with its associated weight $w(u, v)$.

For A* search we need to make use of a heuristic. To save you from dealing with square roots, as we did in the example from class, we will use the L_1 (also called Manhattan or

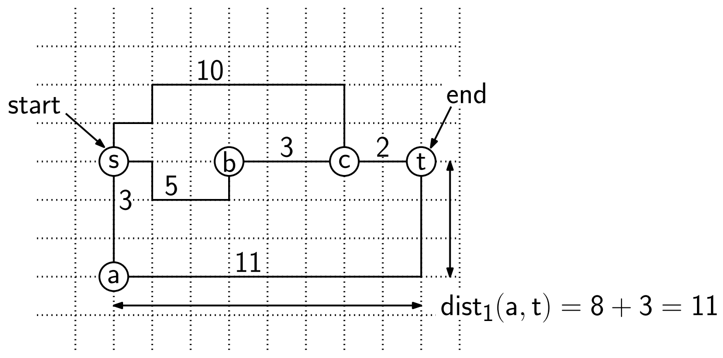


Figure 2: Problem 3.

checkerboard) distance between two points. It is defined to be the sum of the absolute values of the difference of the x and y coordinates of the points. For example, in the figure the L_1 distance between nodes a and t is $\mathrm{dist}_1(a, t) = 8 + 3 = 11$. For A* search define the heuristic value for each node u to be L_1 distance from u to t. For example, $h(a) = 11$. (For this graph it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

(a) Trace the execution of Dijkstra's algorithm on this graph. For each node indicate the following: (1) list the nodes in the order in which they are processed, (2) whenever a node is processed, indicate which of its neighbors have had their d-values modified, when the algorithm terminates (that is, when t is considered for processing), indicate what the final d-values are for all the nodes. If there are ties for which node is to be processed next, select the node that is earliest in alphabetical order. (As an example, see Lecture 19.)

(b) Trace the execution of A* Search on this graph. Provide the same information as in (a), but also provide the A* f-values for each node that is processed (recall that $f(u) = d[u] + h(u)$).

(c) Remark on the differences (if any) in the length of the path generated and the differences (if any) in the efficiency between these two algorithms.

Problem 4. Recall that in visibility-based pursuit-evasion games, you are given a domain and two agents, a pursuer p and an evader e. The pursuer selects a path π through the domain, and moves at constant speed along this path. The evader has knowledge of this path and can predict the exact location of the pursuer at any time. The evader can move at arbitrarily high speed. For a given pursuit path π, the evader *avoids detection* if it is possible for the evader to move in such a way that the pursuer never has a line of sight to the evader. If the pursuer can find a path π for which the evader cannot avoid detection, then the pursuer wins. If for every possible path chosen by the pursuer, the evader can avoid detection, then the evader wins

Consider the three domains shown in Fig. 3. For each, indicate whether the pursuer wins or the evader wins. If the pursuer wins, draw an example of a path π for which the evader cannot avoid detection. (For the fullest credit, your path should be reasonably short, and not involve unnecessary detours.) If the evader wins, give an intuitive explanation as to why the pursuer cannot find such a path. (It is not necessary that your explanation is a rigorous proof. Explain where the "trouble spots" are within the domain.)

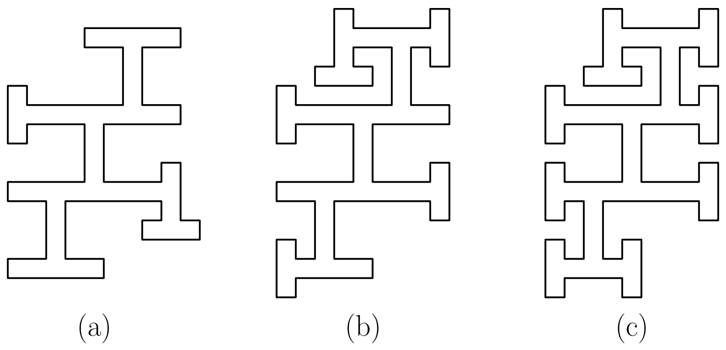


Figure 3: Problem 4.

Problem 5. In this problem we consider motion planning in a dynamic setting. The objective is to design a gamebot that can play a simplified version of the classic video game Galaxian.

You are given a *robot* that consists of a line segment of unit length that resides on the x-axis. The robot can move left or right (but not up or down) at arbitrary speeds. You are given two real values x^- and x^+, and the robot must remain entirely between these two values at all times (see Fig. 4). The robot's *reference point* is its left endpoint, and at time $t = 0$, the left endpoint is located at x^-. (You may assume that $x^+ > x^- + 1$.)

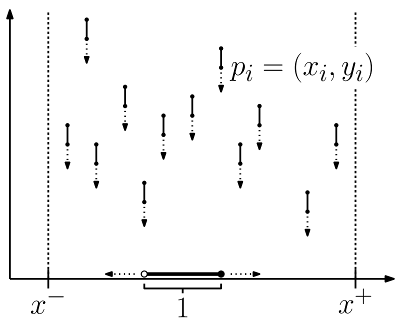


Figure 4: Problem 5.

You are also given a set of *missiles* in the form of n vertical line segments, each of length 0.2, that fall down from the sky at a speed of 2 units per second. Each of these vertical segments is specified by the coordinates of its lower endpoint at time $t = 0$. So, if $p_i = (x_i, y_i)$ is the starting position of the ith missile, then at time t its lower endpoint is located at $(x_i, y_i - 2t)$, and its upper endpoint is at $(x_i, y_i - 2t + 0.2)$. You may assume that $x^- \leq x_i \leq x^+$.

The question is whether it is possible for the robot to move in a manner to avoid all the missiles. We will explore an algorithm for solving this problem.

(a) A natural way to define the robot's configuration at any time is as a pair (t, x), where t is the current time, and x is the x-coordinate of the robot's left endpoint. Based on this, what is the C-obstacle associated with a missile whose starting position is p_i (as defined above)? In other words, describe the set of robot configurations (t, x) such that the robot intersects this missile. (Explain/draw the exact shape of the C-obstacle, its dimensions, and its location in space. Don't just express it abstractly as a Minkowski sum.)

(b) Given the C-obstacles for all the missiles, under what circumstances is it possible to win the game? Express your answer by describing the properties of a path in configuration space, leading from the starting position to the end of the game. (I do not need a complete algorithm, just an explanation of what properties of the C-obstacle placements allow the game to be won.)

(c) Suppose you are told that the robot has a maximum speed σ (in units per second) that it can move at. How would you modify your answer to (b) to address this new limitation?